# Parallel Bootstrap Aggregation with Random Forests

Joshua Hahn jyh2134

December 2022

## 1 Introduction

Bootstrap aggregating, or "bagging", is a powerful tool that helps create accurate, stable, and robust learning models. In this paper, we explore a Random Forest classifier implementation of the bagging algorithm by creating a sequential Haskell implementation, then creating several iterations of the original code using parallelism strategies powered by Haskell's Control.Monad.Par and Control.Parallel.Strategies packages. We evaluate our model on a modified version of Fisher's Iris dataset. Throughout the paper, we will refer to Threadscope reports to evaluate both the speed and parallelism of the code. Section 2 details the various algorithms used in this project. Section 3 details the sequential implementation of the algorithm. Section 4 details the optimizations, parallelisms, and speedups performed.

## 2 Algorithm Overview

### 2.1 Bagging Algorithms

Bootstrap Aggregating, or "bagging" is a machine learning meta-algorithm that aims at producing a model that is more robust and accurate when compared to single-classifier models. Bagging algorithms generate a powerful ensemble classifier that is capable of performing classification tasks: given an unseen example represented as a vector of features, the model makes a prediction on what category the example belongs to.

Bagging algorithms follow a 3-phase process: in the bootstrapping phase of the algorithm, the original dataset is partitioned into bootstrap samples. In the aggregating phase of the algorithm, weak learners independently train on the bootstrap samples. In the final classification stage, a vector is passed to each of the weak learners, and a majority vote is performed to produce the final classification. In this section, we also briefly discuss how each of the steps can be parallelized.

The bootstrapping phase of the algorithm takes a input a training set $D$ of feature-label pairs, where features are represented as vectors of features that describe the example, while the labels denote the species of Iris flower (denoted as category 1 and category 2). The dataset is partitioned into $l$ bootstrap samples, each with size $m$ and labeled $D_i$. The samples are generated by sampling $l$ elements from $D$ with replacement, creating $l$ pseudo-independent samples that each represent a fraction of the original dataset $D$ we are trying to model. This portion of the algorithm is highly parallelizable; since we are taking $m$ independent samples from the dataset $l$ times, each sampling process can be performed in parallel while producing a serializable outcome.

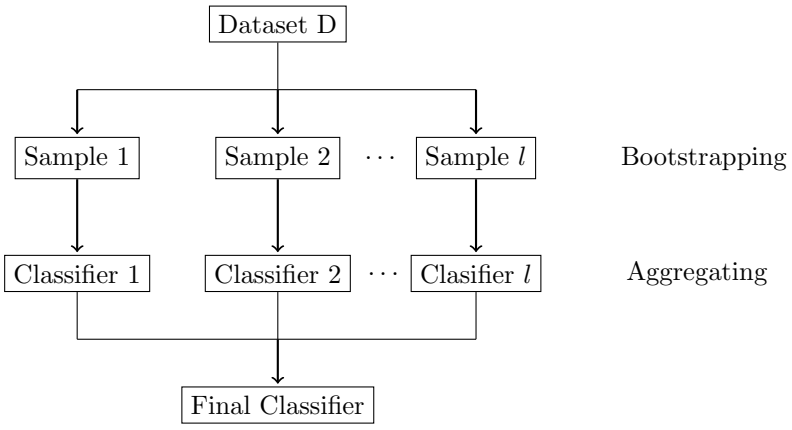



Figure 1: The Bagging Algorithm

The aggregating phase of the algorithm trains independent weak learners on each of the samples, each identifying unique features of the original dataset. This part of the algorithm is also highly parallelizable. Since we are training $l$ independent classifiers on $l$ unique samples, each training protocol can be run in parallel with each other, meaning that the theoretical upper bound of the speedup of this portion is $l$-fold.

Finally, during the classification stage, we take a majority vote of the classes, which means running the test case through each of the classifiers, then seeing how many classifiers return category 1, and seeing how many classifiers return category 2. This process is also parallelizable, since we are running $l$ independent classifiers.

Each step of the bagging algorithm is parallelizable. However, as will be explored below, a majority of the time of this program is spent reading in the dataset $D$. As with all other IO operations, file-reading is sequential by nature, and served as a bottleneck for the speedup of my program.

## 2.2 Random Forests

Random Forest classifiers are a common choice to be used in bagging algorithms, as decision trees tend to be proficient at capturing features of the dataset. By leveraging the power of bagging algorithms and using multiple decision trees to capture unique features of the dataset, we create a classifier that is capable of producing accurate and robust predictions.

Decision trees are trees that follow the format: each TreeNode is either a (Threshold, LeftTree, RightTree) tuple, or a Leaf (Category). For each non-leaf TreeNode, the decision tree creates a split– data values with a feature value that fall below the threshold are categorized into the LeftTree, while data values that fall above the threshold are categorized into the RightTree. Data rows fall through the tree until it reaches a leaf, where it is classified into a category.

The training is done by determining the appropriate thresholds that the Decision Tree should split on, while also controlling how many features each Decision Tree should split based on. For the sake of ensemble algorithms, it is sufficient for the classifiers to each have an accuracy over 50%, since the aggregation of the classifiers in the classification stage of the algorithm does the heavy lifting in converging to a higher classification accuracy. Proof of this claim can be found in the appendix of this paper.

# 3 Sequential Implementation

The Haskell implementation of the Random Forest portion of the algorithm is a bit more simplified. Using domain knowledge of the Iris flower dataset, we know that the features are linearly separable, and that the boundary between the 2 classes of flowers is very clearly defined. As such, we use a primitive random-forest generation algorithm that separates the features by the mean value of the feature values seen in the sample dataset. This approach works well in practice for this domain, but it should be noted that this approach may not generalize well for more complex data sets.

We first begin by defining the core data type for this project, which is the DecisionTree data type. In my implementation of the DecisionTree data type, a node is either a leaf or a Threshold containing two children DecisionTreees. For non-leaf nodes, the threshold value splits the incoming data values into the left decision tree (if the feature specified is less than the threshold), or into the right decision tree (if the feature specified is greater than the threshold).

```haskell
data DecisionTree a b
    = Threshold a Int (DecisionTree a b) (DecisionTree a b)
    | Leaf b
    deriving Show
```

We also define the training algorithm that creates DecisionTrees out of input data (the choice for using V.Vectors as opposd to standard Prelude lists will be discussed in section 4). The tree generates itself recursively level by level. I decided to limit the tree depth to 2, since domain knowledge of the Iris dataset tells us that each example is measured as a vector in $\mathbb{R}^4$, and random forests work best when limited to a depth of $\sqrt{d}$.

```haskell
train :: (V.Vector (V.Vector Double)) -> (Maybe Double) -> [Int] -> (DecisionTree Double Int)
train rows threshold (f1:f2:[]) = case threshold of
    -- Initialize the separator to be the mean of the features.
    Nothing -> train rows (averageOf rows f1) [f1,f2]
    Just x  -> Threshold x f1 (train rows1 Nothing [f2]) (train rows2 Nothing [f2])
        where (rows1, rows2) = rowSplit rows x f1

train rows threshold (f2:[]) = case threshold of
    Nothing -> train rows (averageOf rows f2) [f2]
    Just x  -> Threshold x f2 (train rows1 Nothing []) (train rows2 Nothing [])
        where (rows1, rows2) = rowSplit rows x f2

train rows _ ([]) = Leaf (majority rows)
```

When creating a new DecisionTree, we pass the initial splitting threshold to *Nothing*, which will then be called by the training function and instantiated to a *Just Double* containing the threshold. As stated above, this threshold value will be initialized to the average of the data values we have seen for the specified feature.

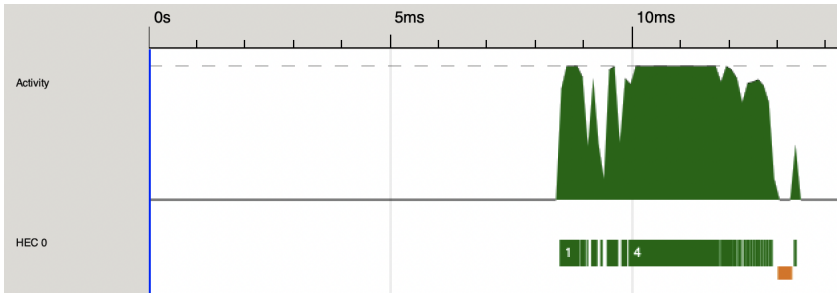
```haskell
-- Calculates the average value of the feature.
averageOf :: V.Vector (V.Vector Double) -> Int -> (Maybe Double)
averageOf rows feature = Just (total / (fromIntegral $ V.length rows))
    where total = V.foldl (\x y -> x + (V.!) y feature) 0 rows
```

We bring the DecisionTree data type and the training function together in the main function, where we first undergo sequential file I/O to input all of the training and testing data, perform the bagging algorithm and generate weak classifiers, then classify each row in the testing set and evaluate against the actual labels. Details of the *cleanup* and the *slice* operations are discussed further in section 4.

```
module Main (main) where

import qualified Data.Vector as V

import DecisionTree
import Control.Parallel.Strategies

main :: IO ()
main = do
    fileName <- return "iris.txt"
    contents <- readFile fileName
    let
        inputData = (V.fromList . cleanup . lines) contents
        training_set = V.slice 0 80 inputData -- Training data is the first 80 rows
        testing_set = V.slice 80 20 inputData -- Testing data is the last 20 rows

        partitionIndices = [0,10,20,30,40,50]
        features = [[a,b] | a <- [0,1,2,3], b <- [0,1,2,3]]
        samples = fmap parMap (partition training_set 30) partitionIndices
        trainingPairs = zip samples features

        forest = fmap (\(s,f) -> train s Nothing f) trainingPairs
        predictions = fmap (\x -> evaluate forest x) testing_set
        solutions = fmap (\x -> (V.!) x 4) testing_set

    putStrLn "Predictions:"
    mapM_ (putStrLn . show) predictions

    putStrLn "Answers:"
    mapM_ (putStrLn . show) solutions
```

Using the given dataset, we find that our accuracy is 100% for all of the rows in the testing set. However, this model was purposefully tested on a highly linearly-separable dataset with clear decision boundaries, and the size of the dataset was also kept very small on purpose to limit the influence of file I/O time on the project. In future iterations of the project, it will be more interesting to work with a more complex dataset, using a more advanced training algorithm, and working with more rows of data.

Figure 2 shows a Threadscope analysis of the eventlog for the sequential implementation of the program. The average runtime for the sequential model is around 0.3 seconds. For the following speedups, this will be used as the baseline.



# 4 Parallelizations & Optimizations

## 4.1 Vectors

In earlier sections, we discussed how the bootstrapping process can be parallelized by performing $l * m$ independent samples from the dataset $D$ with replacement. However, before implementing this portion, I implemented a clever trick to perform a series of independent samples not by individual selections, but by selections in chunks. For this, we use the Vector data defined in the Data.Vector package.

Vectors in Haskell offer a very useful tool: $O(1)$ slice operations. That is, Vectors allow us to generate a subset of the original Vector without copying values in $O(1)$. This offers two very critical benefits for this project: by ensuring that the sliced vectors are not linked to the original vector, we allow each slice to be independent of each other, and therefore, can be run in parallel with other slices simultaneously without having to worrying about having cross-sample contamination. Secondly, $O(1)$ operations means that the parallelization of the bootstrapping process will become even quicker.

To ensure that our project works under the Vector data type as opposed to Prelude lists, we must define a few operations. The first is an operation that can split the CSV file (i.e. the Iris flower dataset) on commas, another operation that can turn the comma-split strings into a list of Vectors containing doubles, and an operation that can perform the aforementioned partition function. The three functions are defined below.

```
-- Creates a partition of the data rows. O(1) operation.
partition :: (V.Vector(V.Vector Double)) -> Int -> Int
    -> (V.Vector(V.Vector Double))
partition rows size start = V.slice start size rows

-- Turns the input string array into a Vector of Vectors of Doubles.
cleanup :: [String] -> [V.Vector(Double)]
cleanup []     = []
cleanup (r:rs) = row:(cleanup rs)
    where row = V.fromList (map (read :: String -> Double) splitR)
          splitR = splitComma r

-- A modification of the prelude 'words' method
splitComma   :: String -> [String]
splitComma s =  case dropWhile (\x -> (x == ',')) s of
                    "" -> []
                    s' -> w : splitComma s''
                        where (w, s'') = break (\x -> (x == ',')) s'
```

## 4.2 ParMap and ParList

ParMap allows us to apply a function to a list, and run the function on each element of the list in parallel. To see where I could apply parMap, I first identified 4 loops where a map was being performed. The first area was when I was partitioning the original dataset into $l$ bootstrap samples, as mentioned above in the slice section. The second area was during the aggregation phase, where I was generating a random forest of $l$ decision trees by mapping over the bootstrap samples. The final area was when I ran the test row through each of the decision trees. The final area is during the data cleanup stage, where we take the input strings and convert them into Vectors of (Vector Double)s.

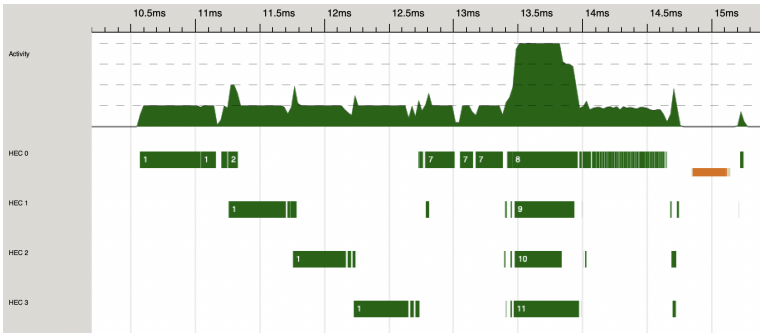
I first wanted to tackle the second loop, where I would parallelize the training step of the program. I predicted that this would be the longest portion of the code (other than file I/O), so I wanted to make sure this part ran as fast as it could.

My first approach to using ParMap was to directly use it to the training function, which would perform the aggregation stage and generate the random forest from the dataset in parallel. However, this soon turned out to be problematic, as parMap could not work on the DecisionTree datatype that I had generated. In particular, *NFData* did not have an instance of the *DecisionTree Double Int* data type that I had constructed, and would not allow me to perform a parMap directly.

Instead, I opted to utilize the *using* keyword, and implemented the *parList* function to iterate over the list of training pairs I had generated. The next decision I had to make was which strategy I would use. In the final iteration of the project, I opted to use the *rpar* strategy, which would spark the evaluation of all of the decision trees in parallel, but not necessarily wait for the evaluation of other sparks (like *rseq*) or to fully evaluate from weak head normal form (like *rdeepseq*). As such, I thought that my decision to use *parList* with *rpar* made sense.

Next, I decided to tackle the first loop, which generated the bootstrap samples from the original dataset. Compared to the second loop, this is a much less computationally expensive process, and I doubted that parallelizing this portion would lead to a large speedup. Moreover, compared to the second loop, implementing *parMap* was much more straightforward: using *parMap* to run the sampling in parallel and then using *runPar* to evaluate the Par monad and retrieve all of the results together, I was able to parallelize the first loop as well.

The following is a Threadscope evaluation of the parallelizations up to this point, using $N4$. The average runtime at this point in time was 0.21 seconds, which is a $1.42x$ speedup from the sequential implementation.




We can see that as expected, file I/O and data cleanup takes up a large portion of the initial program runtime. However, we can also see parallelization happening during the bootstrap and aggregation portions of the algorithm, leaving us with the data cleanup and classification loops to be parallelized.

However, with the sequential file reading portion at the beginning of the program, it is interesting to note the peculiar "passing off" of the IO to different cores at different times. This is further explored in section 5.

For the data cleanup, we note that the previous cleanup method does not work, since it recursively works through the entire dataset, as opposed to one row in the dataset. As such, we introduce a single-line cleanup method.

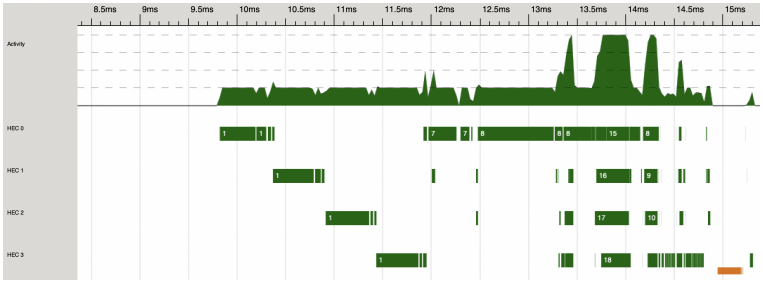
```
cleanLine :: String -> V.Vector(Double)
cleanup r = V.fromList (map (read :: String -> Double) (splitComma r))
```

Using the new cleanLine method, *runPar*, and *parMap*, we are able to achieve parallelism in the cleanup stage as well. Unfortunately, this did not seem to impact runtime at all, since the cleanup was a very small portion of the program to begin with.

Finally, we aim to parallelize the classification stage of the program. For this, we add parallelism to the evaluate method, where we use *parList* and the *rpar* strategy to spark the classifications in parallel, then parallelize the outer loop of the function calls as well using the same process. With all of the desired loops in the program parallelized, we are able to achieve a runtime of 0.161 seconds on 4 cores (with slightly slower runtimes with less or more cores), which is a speedup of $1.86x$ from the baseline. Unfortunately, this is not the most optimal runtime, since a large portion of this program depends on File I/O in the form of reading from a large dataset, and system output in the form of printing all the predictions and the expected values.

```
evaluate :: [DecisionTree Double Int] -> (V.Vector Double) -> Int
evaluate classifiers row
    | ones >= 3 = 1
    | otherwise = 2
    where ones = length $ filter (==1) results
          results = map (classify row) classifiers `using` parList rpar
```

```
forest = map (\(s,f) -> train s Nothing f) trainingPairs `using` parList rseq
predictions = P.runPar $ P.parMap (\x -> evaluate forest x) testing_set
solutions = P.runPar $ P.parMap (\x -> (V.!) x 4) testing_set
```



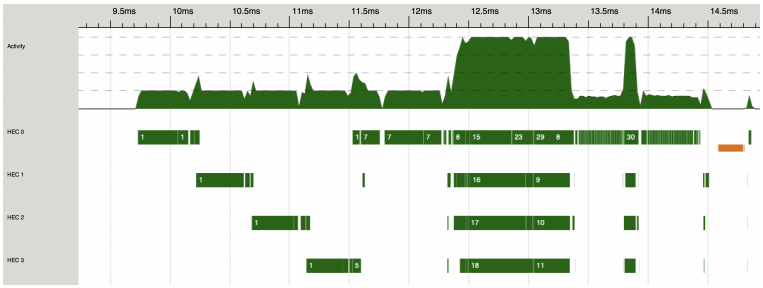

# 5 Further Experiments

In this section, we explore different modifications to the program in hopes of better understanding the peculiarities. Namely, we hope to explore the effect of larger datasets on the runtime and parallelism as suggested by Professor Edwards during my presentation of my project.

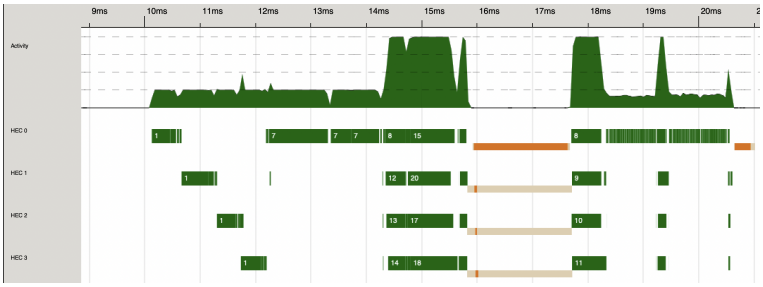
To demonstrate the effects of a larger dataset, we run the program on a second dataset, which contains data of the same format, but with 5 times and 10 times the number of rows (500 and 1000 rows).

For a $5x$ increase in dataset size, we have the following the Threadscope report. Contrary to my fears, it seems like a larger dataset highlights the parallelism that is achievable in this project, rather than be overshadowed by file IO. We see that the middle portion of the program which we aimed to parallelize is much more even and consistent across the cores. However, it

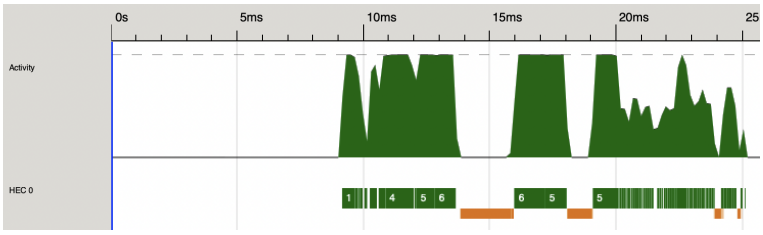
is interesting to note that the four distinct areas of parallelism that I showed in my code is no longer present, and is instead replaced by one large portion of parallelism. I suspect that this is due to the sparks taking much longer to be computed, and residing as threads in the core for longer, sometimes taking more than the program's runtime between the parallel sections to fully compute, which is why they start bleeding into the other areas.




However, when we increase the size of the dataset to 10 times the original program's training set, we see a problem with garbage collection. In the Threadscope report below, we notice the same even parallelism, but it is broken by an unfortunate stalling due to garbage collection. Although I am not sure, I suspect that this is due to the Data.Vector datatype taking up too much space in the memory, where it is competing with the sparks for computation. The Data.Vector data type makes a trade off between fast indexing and slicing with space in memory, and the larger the datasets, the larger the Data.Vector takes up in memory.




We can verify that this is not a product of parallel spark creation but in fact caused by file IO by comparing this Threadscope report to the report generated when this program is run sequentially. As we can see in the report below, the single-threaded sequential execution of the program also suffers from stalling caused by garbage collection. In fact, it seems like the single-threaded version is even worse at GC-ing, as it requires two stalls as opposed to one.



# 6 Conclusion

By identifying four areas of parallelism in the bagging algorithm with random forests, I was able to achieve a speedup of $1.86x$ from the baseline. In future iterations of this project, it will be valuable to work with datasets of greater complexity and aim to optimize I/O operations as a possible area of speedup. Moreover, there were many areas of the project that could have been optimized, including use of more advanced parallelism and perhaps the Repa package as opposed to the vector package.

# 7 Appendix

## 7.1 Proof of convergence for bagging algorithms

Assume we have $l$ bootstrap samples, and we wish to train $l$ classifiers $C_1...C_l$ on the individual samples. Under two assumptions, we can conclude that conducting a majority vote on the classifiers will yield the correct result.

1. Each of the bootstrap samples are independent from each other.
2. Each weak classifier has an accuracy greater than statistical randomness.

We know that for the classifiers' majority vote to be correct, at least $\frac{l}{2}+1$ of the votes must be correct. Assume that each classifier has an accuracy $p$, and $p > 0.5$, assuming we are making a binary classification. (That is, if we were to randomly make N votes, we would make the correct vote $\frac{l+1}{2}$ times, since we are choosing between two classes).

The probability that we will have $\frac{l+1}{2}$ correct votes is $\binom{l}{(\frac{l+1}{2})} * p^{\frac{l+1}{2}} * (1-p)^{l-\frac{l+1}{2}}$, since this follows a binomial distribution of l independent events. Since we can have $n \in [\frac{l+1}{2}, l]$ correct votes and still produce the correct answer, we sum from $\frac{l+1}{2}$ to $l$ to get a function of accuracy.

Let $f(p,l) = \Sigma_{n=\frac{l+1}{2}}^{l} \binom{l}{n} * p^n * (1-p)^{l-n}$. Plotting $f$ against $l$ at varying fixed $p$, we have:

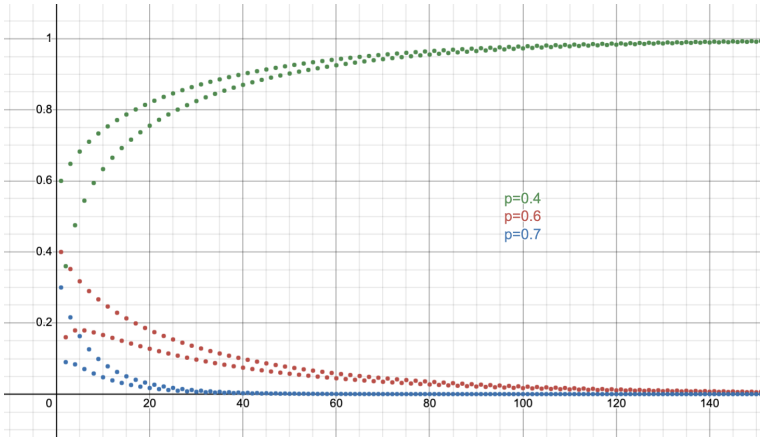



Figure 2: The model's accuracy across various p-values

## 7.2 Code Listing

**Main.hs**

```
module Main (main) where

import qualified Data.Vector as V

import DecisionTree
import Control.Parallel.Strategies
import Control.Monad.Par as P

main :: IO ()
main = do
    fileName <- return "iris.txt"
    contents <- readFile fileName
    let
        contentLines = lines contents
        cleanLines = map cleanLine contentLines `using` parList rpar
        inputData = V.fromList cleanLines

        training_set = V.slice 0 80 inputData -- Training data is the first 80 rows
        testing_set = V.slice 80 20 inputData -- Testing data is the last 20 rows

        partitionIndices = [0,10,20,30,40,50]
        features = [[a,b] | a <- [0,1,2,3], b <- [0,1,2,3], a /= b]
        samples = P.runPar $ P.parMap (partition training_set 30) partitionIndices
        trainingPairs = zip samples features

        forest = map (\(s,f) -> train s Nothing f) trainingPairs `using` parList rseq
        predictions = P.runPar $ P.parMap (\x -> evaluate forest x) testing_set
        solutions = P.runPar $ P.parMap (\x -> (V.!) x 4) testing_set

    putStrLn "Predictions:"
    mapM_ (putStrLn . show) predictions
    putStrLn "Answers:"
    mapM_ (putStrLn . show) solutions

    -- Now we have to evaluate the forest.
    putStrLn "Done."


-- Creates a partition of the data rows. O(1) operation.
partition :: (V.Vector(V.Vector Double)) -> Int -> Int
    -> (V.Vector(V.Vector Double))
partition rows size start = V.slice start size rows

-- Turns the input string array into a Vector of Vectors of Doubles.
cleanup :: [String] -> [V.Vector(Double)]
cleanup []     = []
```

```
cleanup (r:rs) = row:(cleanup rs)
    where row = V.fromList (map (read :: String -> Double) splitR)
          splitR = splitComma r

cleanLine :: String -> V.Vector(Double)
cleanLine r = V.fromList (map (read :: String -> Double) (splitComma r))

-- A modification of the prelude 'words' method
splitComma   :: String -> [String]
splitComma s =  case dropWhile (\x -> (x == ',')) s of
                     "" -> []
                     s' -> w : splitComma s''
                           where (w, s'') = break (\x -> (x == ',')) s'
```

**DecisionTree.hs**

```
module DecisionTree where

import qualified Data.Vector as V
import Control.Monad.Par as P

data DecisionTree a b
    = Threshold a Int (DecisionTree a b) (DecisionTree a b)
    | Leaf b
    deriving Show

train :: (V.Vector (V.Vector Double)) -> (Maybe Double) -> [Int] -> (DecisionTree Double Int)
train rows threshold (f1:f2:[]) = case threshold of
    -- Initialize the separator to be the mean of the features.
    Nothing -> train rows (averageOf rows f1) [f1,f2]
    Just x  -> Threshold x f1 (train rows1 Nothing [f2]) (train rows2 Nothing [f2])
        where (rows1, rows2) = rowSplit rows x f1

train rows threshold (f2:[]) = case threshold of
    Nothing -> train rows (averageOf rows f2) [f2]
    Just x  -> Threshold x f2 (train rows1 Nothing []) (train rows2 Nothing [])
        where (rows1, rows2) = rowSplit rows x f2

train rows _ ([]) = Leaf (majority rows)

classify :: (V.Vector Double) -> (DecisionTree Double Int) -> Int
classify row classifier = case classifier of
    Threshold val feature leftTree rightTree
        | ((V.!) row feature) < val -> classify row leftTree
        | otherwise                 -> classify row rightTree
    Leaf category                   -> category

evaluate :: [DecisionTree Double Int] -> (V.Vector Double) -> Int
evaluate classifiers row
    | ones >= 3 = 1
```

```
    | otherwise = 2
    where ones = length $ filter (==1) results
          results = map (classify row) classifiers `using` parList rpar

-- Decides the majority vote on the feature.
majority :: (V.Vector (V.Vector Double)) -> Int
majority rows
    | 2 * ones > V.length rows = 1
    | otherwise           = 2
    where
        ones = V.length $ (V.filter (\x -> ((V.!) x 4) == 1)) rows

-- Splits the dataset on the threshold and feature.
rowSplit :: (V.Vector (V.Vector Double)) -> Double -> Int
    -> ((V.Vector (V.Vector Double)),  (V.Vector (V.Vector Double)))
rowSplit rows threshold feature = (r1, r2)
    where
        r1 = V.filter (\x -> ((V.!) x feature) < threshold) rows
        r2 = V.filter (\x -> ((V.!) x feature) >= threshold) rows

-- Calculates the average value of the feature.
averageOf :: V.Vector (V.Vector Double) -> Int -> (Maybe Double)
averageOf rows feature = Just (total / (fromIntegral $ V.length rows))
    where total = V.foldl (\x y -> x + (V.!) y feature) 0 rows
```